Dynamical Systems CW5

CID: 02044064

March 14, 2025

(a)-(b) Parameter Search

I tried grid search, gradient descent with momentum (GDM) and Newton's method. In general, GDM and Newton's method can get stuck in local minima, but in our case, there is a unique global minimum, as can be seen in section (c), so for both algorithms, I only tried a single starting point to find the global minimum. The momentum in gradient descent means that the algorithm picks up speed (by taking larger strides) as it rolls down a "hill" on the error surface. This will be explained in more detail in section (d). The code for each algorithm is shown below. My code is largely based on lecture 12.

Grid Search vs GDM

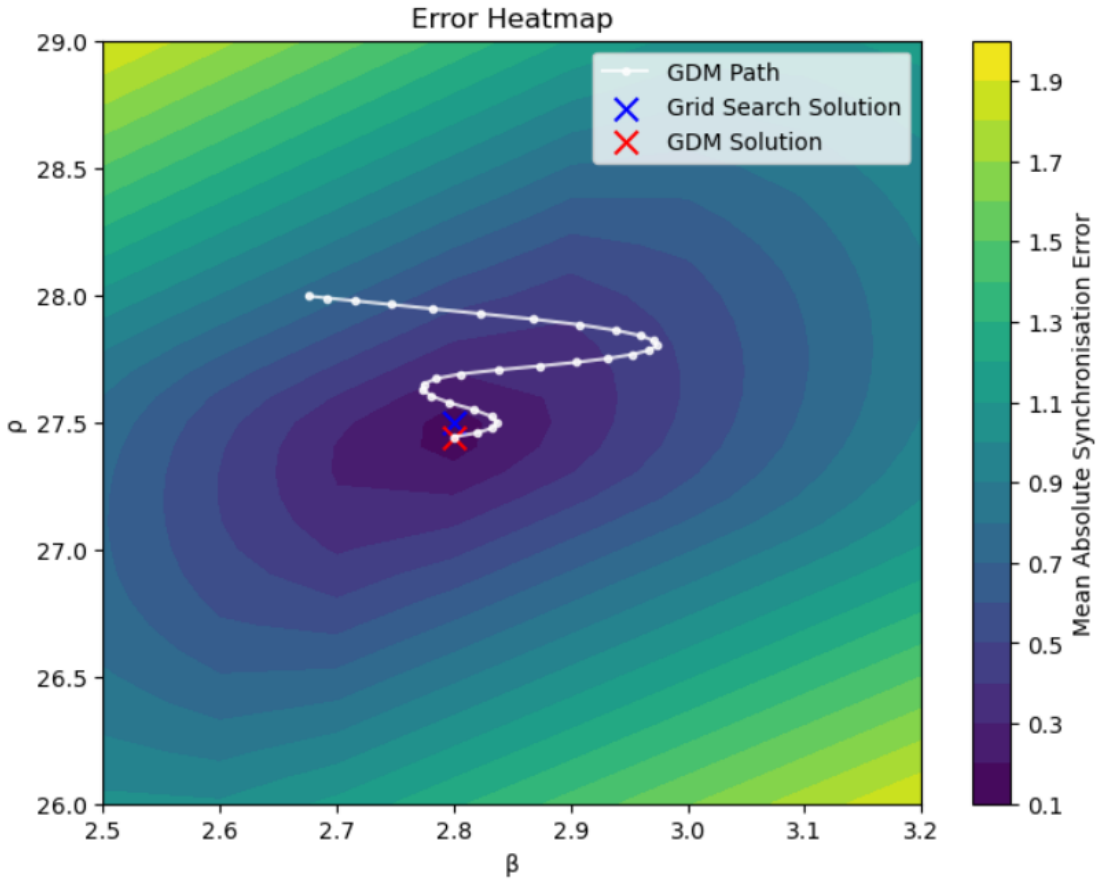
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from tqdm import tqdm  # Progress tracking

# Load data
data = np.loadtxt("lorenz_xz_data.txt")
t_data, x_data, z_data = data[:, 0], data[:, 1], data[:, 2]

def driven_lorenz(t, state, rho, beta):
    y, z = state
    x = np.interp(t, t_data, x_data)
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return [dy, dz]

def simulate_lorenz(rho, beta):
    y0, z0 = 0, 1  # Initial conditions
    sol = solve_ivp(driven_lorenz, (t_data[0], t_data[-1]), [y0, z0], t_eval=t_data,
    ↪  args=(rho, beta), method='RK45')
    return sol.y[1]  # Return z(t) values

def mean_absolute_sync_error(rho, beta):
    z_sim = simulate_lorenz(rho, beta)
    return np.mean(np.abs(z_sim - z_data))

# Grid search parameter space
rho_values = np.arange(26, 29.001, 0.1)
beta_values = np.arange(2.5, 3.201, 0.1)
errors = np.zeros((len(rho_values), len(beta_values)))

for i, rho in enumerate(tqdm(rho_values, desc="Running Grid Search")):
```

```
    for j, beta in enumerate(beta_values):
        errors[i, j] = mean_absolute_sync_error(rho, beta)

# Optimal parameters from grid search
min_idx = np.unravel_index(np.argmin(errors), errors.shape)
optimal_rho_grid = rho_values[min_idx[0]]  # Access the optimal rho value
optimal_beta_grid = beta_values[min_idx[1]]  # Access the optimal beta value
optimal_mase_grid = errors[min_idx]

# Gradient descent with momentum
def gradient_descent_momentum(learning_rate = 0.05, momentum=0.9, max_iterations=30,
↪  tol=1e-5):
    rho, beta = 28, 8/3  # Initial guess
    explored_params = []
    prev_loss = np.inf
    v_rho, v_beta = 0, 0  # Initialise velocity terms

    for _ in tqdm(range(max_iterations), desc="Running Gradient Descent with Momentum"):
        delta = 0.01
        grad_rho = (mean_absolute_sync_error(rho + delta, beta) -
        ↪  mean_absolute_sync_error(rho - delta, beta)) / (2 * delta)
        grad_beta = (mean_absolute_sync_error(rho, beta + delta) -
        ↪  mean_absolute_sync_error(rho, beta - delta)) / (2 * delta)

        # Update velocities
        v_rho = momentum * v_rho + (1-momentum) * grad_rho
        v_beta = momentum * v_beta + (1-momentum) * grad_beta

        # Apply updates
        rho -= learning_rate * v_rho
        beta -= learning_rate * v_beta

        current_loss = mean_absolute_sync_error(rho, beta)
        explored_params.append((rho, beta, current_loss))

        if np.abs(prev_loss - current_loss) < tol:
            break
        prev_loss = current_loss

        rho = np.clip(rho, 26, 29)
        beta = np.clip(beta, 2.5, 3.2)

    return rho, beta, np.array(explored_params)

# Run gradient descent with momentum
rho_best_gdm, beta_best_gdm, explored_params = gradient_descent_momentum()

# Extract trajectory for plotting
rho_traj, beta_traj = explored_params[:, 0], explored_params[:, 1]

# Plot heatmap
plt.figure(figsize=(8, 6))
contour = plt.contourf(beta_values, rho_values, errors, levels=20, cmap='viridis')
plt.colorbar(contour, label="Mean Absolute Synchronisation Error")
```

```
83
84  # Overlay GDM trajectory
85  plt.plot(beta_traj, rho_traj, marker='o', linestyle='-', color='white', markersize=3,
    ↪  alpha=0.8, label="GDM Path")
86
87  # Mark optimal points
88  plt.scatter(optimal_beta_grid, optimal_rho_grid, color='blue', marker='x', s=100,
    ↪  label="Grid Search Solution")
89  plt.scatter(beta_best_gdm, rho_best_gdm, color='red', marker='x', s=100, label="GDM
    ↪  Solution")
90
91  plt.xlabel("")
92  plt.ylabel("")
93  plt.title("Error Heatmap")
94  plt.legend()
95  plt.show()
96
97  # Print solutions
98  print(f"Optimal parameters from grid search:  = {optimal_rho_grid:.4f},  =
    ↪  {optimal_beta_grid:.4f}, Error = {optimal_mase_grid:.6f}")
99  print(f"Optimal parameters from GDM:  = {rho_best_gdm:.4f},  = {beta_best_gdm:.4f}, Error
    ↪  = {mean_absolute_sync_error(rho_best_gdm, beta_best_gdm):.6f}")
```

(c) Data Visualisation

Grid Search vs GDM



Potential sources of error

My gradient descent algorithm computes gradients numerically via finite differences rather than analytically (since our error metric involves the modulus function which is not differentiable everywhere, an analytical expression might be invalid for certain parameter values). As a result, the gradient computation is only an approximation to the true gradient.

solve_ivp is another potential source of error because it solves the differential equation numerically, so it cannot recover the true trajectory of z. This means that the loss function in gradient descent is not entirely accurate, so even if our gradient descent algorithm found the exact parameter values of the global minimum, these will not be the true parameter values, as it is not the global minimum of the true loss function.

Limitations

As mentioned earlier, gradient descent might not find the global minimum (especially if the loss landscape was highly non-convex), because the algorithm I used does not take any random steps that can get it out of a local minimum. However, gradient descent with momentum can overcome this issue (intuitively, the algorithm is a ball that picks up momentum as it rolls down a slope which may be enough to keep it rolling uphill for some time).

Compared to gradient descent, Newton's method is computationally inefficient, because if we are estimating N parameters, N values are required to compute the gradient vector as it consists of N first-order derivatives, whereas N^2 values are required for the second-order derivatives, as the Hessian is an N x N matrix, as shown in the following section.

(d) Parameter estimates

I initially tried grid search with grid size of 0.05 x 0.05 as it is more thorough than gradient descent which only explores a subset of the parameter space, and as a result, can get stuck in local minima. On the other hand, grid search scans the entire parameter space and is therefore guaranteed to find the global minimum (approximately, as grid search has fixed resolution). Since the increment size was 0.05, the total number of values explored is (29-26)/0.05 x (3.2-2.5)/0.05 = 840, whereas I limited GDM to at most 30 iterations (stopping early if the absolute value of the difference in loss between two successive iterations falls below a specified tolerance level, such as 10^{-5}) and consequently, at most 30 different parameter pairs. For considerably fewer iterations, the parameters found by gradient descent are very similar to those found by grid search, so gradient descent found the global minimum as well, and it achieves a similar synchronisation error to grid search (see the table on the next page). Although grid search does not require the loss function to be differentiable, its resolution is constant across all parameter values (as the grid is evenly spaced), whereas in gradient descent (with momentum), the "resolution" is variable because the parameter updates become finer as the algorithm approaches a minimum: if μ is the momentum parameter, γ is the learning rate, $\theta = (\beta, \rho)$ and $v = (v_\beta, v_\rho)$, then the GDM update rule can be expressed as

$$v_{t+1} = \mu v_t + (1 - \mu)\nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t - \gamma v_{t+1}$$

where L is the loss function, which is the mean absolute synchronisation error in our case. If θ^* is the global minimum, then as $\theta \to \theta^*$ (and $t \to \infty$),

$$\nabla L(\theta) \to \nabla L(\theta^*) = 0$$

and so

$$\Delta\theta_t = -\gamma \nabla L(\theta_t) \to 0$$

which is precisely what we want as our algorithm approaches the global minimum θ^*: reducing the parameter update size $\Delta\theta_t$ is essentially the same as increasing the resolution around θ^* on a grid search.

Like gradient descent, Newton's method is another iterative scheme, but it looks at the curvature of the loss function (using second-order derivatives) to take a more direct route to the optimum. It can achieve very similar parameter values and error with only 15 iterations. Newton's method is defined by the following update rule:

$$\theta_{t+1} = \theta_t - \gamma[\nabla^2 L(\theta_t)]^{-1} \nabla L(\theta_t)$$

The parameter estimates produced by grid search and GDM are tabulated below.

Method	β	ρ	E
Grid Search	2.8000	27.5000	0.156660
GDM	2.8004	27.4422	0.151994

As GDM yields the lowest error, my final parameter estimates are $(\beta, \rho) = (2.8004, 27.4422)$.